Unidad 04: Tipos, Operadores, y Expresiones

4

Contenidos analíticos

Parte II: Constructos del Lenguaje: Expresiones, Declaraciones, Sentencias y Estructura de Programa

Unidad 3: Tipos, Operadores, y Expresiones
Tipo, precedencia, asociatividad, orden de evaluación. Declaraciones. Definiciones, Alcance. Identificadores, Constantes, Variables, valorL

Tipos de datos

Los tipos de datos Identifican o determinan un dominio de valores y un conjunto de operaciones aplicables sobre esos valores.
Retomamos aquí algunos conceptos de la unidad anterior para profundizar sobre ellos

Tipos de datos simples – Tamaño y Rango

		Implementaciones					
Nombre del Tipo	Otros Nombres	**Borland Turbo C++ 3.0** *16 bits*		**Borland C++ 4.02** *32 bits*		**Microsoft Visual Studio .NET** *32 bits*	
		Bytes	Rango	Bytes	Rango	Bytes	Rango
Char	signed char	1	-128 .. 127 [-2^7 .. (2^7-1)] (ASCII Standard)				
unsigned char	-	1	0 .. 255 [0 .. (2^8-1)] (ASCII Extendido)				
Short	short int, signed short, signed short int	2	–32,768 .. 32,767 [-2^{15} .. (2^{15}-1)]				
unsigned short	unsigned short int	2	0 .. 65,535 [0 .. (2^{16}-1)]				
Int	Signed, signed int	2	–32,768 .. 32,767 [-2^{15} .. (2^{15}-1)]	4	–2,147,483,648 .. 2,147,483,647 [-2^{31} .. (2^{31}-1)]		
unsigned int	unsigned	2	0 .. 65,535 [0 .. (2^{16}-1)]	4	0 .. 4,294,967,295 [0.. (2^{32}1)]		
Long	long int, signed long, signed long int	4	–2,147,483,648 .. 2,147,483,647				
unsigned long	unsigned long int	4	0 .. 4,294,967,295				
Enum	-	*igual a int*					
Float	-	4	3.4 x 10^{-38} .. 3.4 x 10^{38} (7 dígitos de precisión)				
Double	-	8	1.7 x 10^{-308} .. 1.7 x 10^{308} (15 dígitos)				
long double	-	10	3.4 x 10^{-4932} .. 1.1 x 10^{4932} (19 dígitos)			*igual a double*	

Precedencia y Asociatividad de Operadores elementales

ID Asociatividad Izquierda-Derecha, DI Asociatividad Derecha-Izquierda

1 ID	operadores de acceso	( )	invocación a función
		++	pos-incremento
		--	Pos-decremento
12 DI	operadores unarios (operan sobre un solo operando)	+ -	signos positivo y negativo
		!	NOT lógico
		++	pre-incremento
		--	pre-decremento
		sizeof	tamaño de
3 ID	operadores multiplicativos	*	multiplicación
		/	cociente
		%	módulo o resto
4 ID	operadores aditivos	+ -	suma y resta
6 ID	operadores relacionales	< >	
		<= >=	
7 ID	operadores de igualdad	== !=	igual a y distinto de
11 ID	operadores binarios lógicos	&&	AND
12 ID		||	OR
13 DI	operador condicional	? :	(opera sobre 3 operandos)
14 DI	operadores de asignación	=	
		*= /= %= += -=	
		<<= >>= &= ^= |=	
15 ID	operador concatenación expresiones	,	"coma"

- Los operadores **&&** , **||** y **,** "coma" son los únicos que garantizan que los operandos sean evaluados en un orden determinado (de izquierda a derecha).

- El operador condicional (**? :**) evalúa solo un operando, entre el 2do. y el 3ro., según corresponda.

Operadores de C[1]

Símbolo [1]	Tipo de operación	asociatividad
`[ ] ( ) . ->` `++--` (postfijo)	Expresión	De izquierda a derecha
`sizeof & * + - ~ !` `++--` (prefijo)	Unario	De derecha a izquierda
typecasts	Unario	De derecha a izquierda
`* / %`	Multiplicativo	De izquierda a derecha
`+ -`	Aditivo	De izquierda a derecha
`<< >>`	Desplazamiento bit a bit	De izquierda a derecha
`< > <= >=`	Relacional	De izquierda a derecha
`== !=`	Igualdad	De izquierda a derecha
`&`	AND bit a bit	De izquierda a derecha
`^`	OR exclusivo bit a bit	De izquierda a derecha
`|`	OR inclusivo bit a bit	De izquierda a derecha
`&&`	AND lógico	De izquierda a derecha
`||`	OR lógico	De izquierda a derecha
`? :`	Expresión condicional	De derecha a izquierda
`= *= /= %=` `+= -= <<= >>= &=` `^= |=`	Asignación simple y compuesta [2]	De derecha a izquierda
`,`	Evaluación secuencial	De izquierda a derecha

[1] Los operadores se enumeran por prioridad, de mayor a menor. Si aparecen varios operadores en la misma línea o en un grupo, tienen la misma prioridad.

[2] Todos los operadores de asignación simples y compuestos tienen la misma prioridad.

Una expresión puede contener varios operadores con la misma prioridad. Cuando varios operadores de este tipo aparecen en el mismo nivel en una expresión, la evaluación continúa según la asociatividad del operador, de derecha a izquierda y de izquierda a derecha. La dirección de evaluación no afecta a los resultados de las expresiones que incluyen más de un operador de multiplicación (`*`), adición (`+`) o binario bit a bit (`&`, `|` o `^`) en el mismo nivel. El orden de las operaciones no lo define el lenguaje. El compilador es libre de evaluar estas expresiones en cualquier orden, si puede garantizar un resultado coherente.

[1] https://learn.microsoft.com/es-es/cpp/c-language/precedence-and-order-of-evaluation?view=msvc-170

Solo los operadores de evaluación secuencial (`,`), AND lógico (`&&`), OR lógico (`||`), expresión condicional (`? :`) y llamada a función constituyen puntos de secuencia y, por lo tanto, garantizan un orden de evaluación concreto para sus operandos. El operador de llamada a función es el conjunto de paréntesis que siguen al identificador de función. Está garantizado que el operador de evaluación secuencial (`,`) evalúa sus operandos de izquierda a derecha. (El operador de coma en una llamada a función no es igual que el operador de evaluación secuencial y no proporciona esta garantía). Para obtener más información, si lo desea vea puntos de secuencia

Entre "puntos de secuencia" consecutivos, una expresión solo puede modificar una vez el valor de un objeto. El lenguaje C define los puntos de secuencia siguientes:

- Operando izquierdo del operador AND lógico (**&&**). El operando izquierdo del operador AND lógico se evalúa totalmente y todos los efectos secundarios se completan antes de continuar. Si el operando izquierdo se evalúa como false (0), el otro operando no se evalúa.
- Operando izquierdo del operador OR lógico (`||`). El operando izquierdo del operador OR lógico se evalúa totalmente y todos los efectos secundarios se completan antes de continuar. Si el operando izquierdo se evalúa como true (distinto de cero), el otro operando no se evalúa.
- Operando izquierdo del operador de coma. El operando izquierdo del operador de coma se evalúa totalmente y todos los efectos secundarios se completan antes de continuar. Los dos operandos del operador de coma se evalúan siempre. Observe que el operador de coma en una llamada de función no garantiza un orden de evaluación.
- Operador de llamada de función. Todos los argumentos de una función se evalúan y todos los efectos secundarios se completan antes de la entrada a la función. No se especifica ningún orden de evaluación entre los argumentos.
- Primer operando del operador condicional. El primer operando del operador condicional se evalúa totalmente y todos los efectos secundarios se completan antes de continuar.
- El final de una expresión completa de inicialización (es decir, una expresión que no forma parte de otra expresión tal como el final de una inicialización en una instrucción de declaración).
- La expresión de una instrucción de expresión. Las instrucciones de expresión constan de una expresión opcional seguida de un punto y

coma (**;**). La expresión se evalúa para sus efectos secundarios y hay un punto de secuencia después de esta evaluación.
- La expresión de control de una instrucción de selección (**if** o **switch**). La expresión se evalúa completamente y todos los efectos secundarios se completan antes de que se ejecute el código dependiente de la selección.
- La expresión de control de una instrucción **while** o **do** . La expresión se evalúa completamente y todos los efectos secundarios se completan antes de que se ejecute ninguna instrucción de la siguiente iteración del bucle **while** o **do** .
- Cada una de las tres expresiones de una instrucción **for** . Las expresiones se evalúan completamente y todos los efectos secundarios se completan antes de que se ejecute ninguna instrucción de la siguiente iteración del bucle **for** .
- La expresión en una instrucción **return**. La expresión se evalúa completamente y todos los efectos secundarios se completan antes de devolver el control a la función de llamada.

Los operadores lógicos también garantizan la evaluación de sus operandos de izquierda a derecha. Sin embargo, evalúan el número más pequeño de operandos necesarios para determinar el resultado de la expresión. Esto se denomina evaluación de "cortocircuito". Por tanto, es posible que algunos operandos de la expresión no se evalúen. Por ejemplo, en la expresión

```
x && y++
```

se evalúa el segundo operando, `y++`, solo si `x` es true (distinto de cero). Así, `y` no se incrementa si `x` es false (0).

Ejemplos

La lista siguiente muestra cómo el compilador enlaza varias expresiones de ejemplo automáticamente:

Expresión	Enlace automático
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c)`
`q && r || s--`	`(q && r) || s--`

En la primera expresión, el operador AND bit a bit (`&`) tiene prioridad sobre el operador OR lógico (`||`), por lo que `a & b` forma el primer operando de la operación OR lógica.

En la segunda expresión, el operador OR lógico (`||`) tiene una mayor prioridad que el operador de asignación simple (`=`), por lo que `b || c` se agrupa como el operando derecho de la asignación. Observe que el valor asignado a `a` es 0 o 1.

La tercera expresión muestra una expresión correctamente formada que puede generar un resultado inesperado. El operador AND lógico (`&&`) tiene una mayor prioridad que el operador OR lógico (`||`), por lo que `q && r` se agrupa como operando. Puesto que los operadores lógicos garantizan la evaluación de los operandos de izquierda a derecha, `q && r` se evalúa antes de `s--`. En cambio, si `q && r` se evalúa como un valor distinto de cero, `s--` no se evalúa y `s` no se reduce. Si la no reducción de `s` puede provocar un problema en el programa, `s--` debe aparecer como el primer operando de la expresión, o bien `s` se debería reducir en una operación independiente.

La siguiente expresión no es válida y muestra un mensaje de diagnóstico en tiempo de compilación:

Expresión no válida	**Agrupación predeterminada**
`p == 0 ? p += 1: p += 2`	`( p == 0 ? p += 1 : p ) += 2`

En esta expresión, el operador de igualdad (`==`) tiene una prioridad superior, por lo que `p == 0` se agrupa como operando. El operador de expresión condicional (`? :`) tiene la siguiente prioridad más alta. Su primer operando es `p == 0` y su segundo operando es `p += 1`. Sin embargo, el último operando del operador de expresión condicional se considera `p` en lugar de `p += 2`, puesto que esta instancia de `p` se enlaza de una forma más directa con el operador de expresión condicional que con el operador de asignación compuesta. Se produce un error de sintaxis, porque `+= 2` no tiene un operando izquierdo. Debe utilizar paréntesis para evitar errores de esta clase y que el código sea más legible. Por ejemplo, podría utilizar paréntesis, como se muestra a continuación, para corregir y aclarar el ejemplo anterior:

```
( p == 0 ) ? ( p += 1 ) : ( p += 2 )
```

Análisis y ejemplos de uso

El resultado de la división de dos enteros es un entero, la división de enteros es, entonces una división entera, por tanto siendo:

Int a=10, b=2, c=3;
float d = 3.0, e=10.0;

Determine los resultados de

a/b
a/c
a/3.0
a/d
e/d

Justifique la respuesta→ Ayuda: determine que es jerarquía de datos, cual es la secuencia (char, int, long, float, double), que pasa en una expresión con datos de diferentes jerarquías, busque la definición casteo explicito, implicito

Es posible asignar a=b=c=15; que valores asigna a que identificadores, porque, es la signacion = un operador o una sentencia?

Siendo a=10; que efecto produce a++; y si hubiera sido ++a? que hubiera pasado con –
Que efecto produce d++;
Siendo int a = 10, b;
¿Que valores toman a y b luego de b=a++;? ¿Por qué?

Siendo int a = 10, b;
¿Que valores toman a y b luego de b = ++a?; ¿Por qué?

Que valor toman a y b luego de a+=b; y a*=b, Por qué?

Sea int a;
Es posible la asignación a = ‘A’+’B’; esta sentencia le asigna valor al identificador a?, ¿que valor? ¿Por que?

Declaraciones de variables simples

La declaración de una variable simple, la forma más sencilla de un declarador directo, especifica el nombre y el tipo de la variable. También especifica la clase de almacenamiento y el tipo de datos de la variable.

Puede utilizar una lista de identificadores separados por comas (**,**) para especificar varias variables en la misma declaración.

```
int x, y;         /* Declara dos variables simples de tipo int */
int const z = 1; /* Declares declara una constante z de tipo entero y valor 1 */
```

Las variables `x` y `y` pueden contener cualquier valor del conjunto definido por el tipo **`int`** para una implementación concreta. El objeto simple `z` se inicializa con el valor 1 y no es modificable.

Los identificadores x e y pueden ser modificados por lo que pueden estar a la izquierda en una expresión de asignación, por esta razón se los denomina ValorL (left). Esto no ocurre con las constantes. Como veremos mas adelante los argumentos vinculados con parámetros pasados por referencia deben ser valor, los pasados por valor no cecesariamente, admiten expresiones

Si la declaración de `z` fuera para una variable estática no inicializada o estuviera en el ámbito de archivo, recibiría un valor inicial de 0 y ese valor no se podría modificar.

```
unsigned long reply, flag; /* Declara dos variables   */
```

En este ejemplo, ambas variables, `reply` y `flag`, son de tipo **`unsigned long`** y tienen valores enteros sin signo.

Almacenamiento de tipos básicos

En la tabla siguiente se resume el almacenamiento asociado a cada tipo básico.

Tamaños de los tipos fundamentales

Tipo	Almacenamiento
`char`, **`unsigned char`**, **`signed char`**	1 byte
`short`, **`unsigned short`**	2 bytes
`int`, **`unsigned int`**	4 bytes
`long`, **`unsigned long`**	4 bytes
`long long`, **`unsigned long long`**	8 bytes
`Float`	4 bytes
`Double`	8 bytes
`long double`	8 bytes

Los tipos de datos de C entran en categorías generales. Los *tipos enteros* incluyen **`int`**, **`char`**, **`short`**, **`long`** y **`long long`**. Estos tipos se pueden calificar con **`signed`** o **`unsigned`** , y **`unsigned`** solo se puede usar como una abreviatura para **`unsigned int`** . Los tipos de enumeración (**`enum`**) también se tratan como tipos enteros para la mayoría de los propósitos. Los *tipos de punto flotante* incluyen **`float`**, **`double`** y **`long double`**. Los *tipos aritméticos* incluyen todos los tipos flotantes y enteros.

Inicialización

Un "inicializador" es un valor o una secuencia de valores que se deben asignar a la variable que se declara. Se puede establecer una variable en un valor inicial si se aplica un inicializador al declarador en la declaración de variable. El valor o los valores del inicializador se asignan a la variable. En las secciones siguientes se describe cómo inicializar las variables de los tipos escalar, agregado y cadena. Los "tipos escalares" incluyen todos los tipos aritméticos, más los punteros. Los "tipos globales" incluyen matrices, estructuras y uniones.

Inicializar tipos escalares

Al inicializar tipos escalares, el valor de expresión de asignacion se asigna a la variable.

Puede inicializar variables de cualquier tipo, siempre que observe las reglas siguientes:
Las variables declaradas en el nivel de ámbito de archivo se pueden inicializar.
Si no inicializa explícitamente una variable en el nivel externo, se inicializa en 0 de forma predeterminada.
Una expresión constante se puede utilizar para inicializar una variable global declarada con el especificador static storage-class-specifier .
Las variables declaradas como static se inicializan cuando comienza la ejecución del programa.
Si no inicializa explícitamente una variable global static , se inicializa en 0 de forma predeterminada
Las variables declaradas con el especificador de clase de almacenamiento auto o register se inicializan cada vez que el control en tiempo de ejecución pasa al bloque donde se declaran.
Si omite un inicializador en la declaración de una variable auto o register , el valor inicial de la variable es indefinido.
Para los valores auto y register, el inicializador no se limita a una constante; puede ser cualquier expresión que contenga valores definidos previamente, incluidas llamadas de función.
Los valores iniciales de las declaraciones de variables externas y de todas las variables static , tanto sin son externas como internas, deben ser expresiones constantes. (Para obtener más información, vea Expresiones constantes). Como la dirección de cualquier variable definida externamente o variable estática es constante, se puede utilizar para inicializar una variable de puntero static declarada internamente. Sin embargo, la dirección de una variable auto no puede utilizarse como inicializador estático porque puede ser diferente en cada ejecución del bloque. Puede utilizar valores constantes o variables para inicializar las variables auto o register . Si la declaración de un identificador tiene ámbito de bloque y el identificador tiene vinculación externa, la declaración no puede tener una inicialización.

Alcance y visibilidad de los identificadores

Esta propiedad refiere a su visibilidad o “reconocimiento” dentro de la aplicacion. Puede ser un alcance global, que es reconocida en toda la aplicación o local propia de cada modulo o, inclusive, bloque en el que esta
Profundizaremos este tema cuando avancemos en el lenguaje de programación definido para las implementaciones en esta materia